Commodore
International Services Corporation

1200 Wilson Drive
West Chester, PA 19380
(215) 431-9100

March 10, 1992

Dear Sir,

Please find attached a new set of tools disks for CDTV development. As part of your CDTV license agreement, Commodore provides you with utility software, mastering software, and tools to facilitate the creation of CDTV titles. The attached disks represent Commodore's fulfillment of that agreement.

Here follows a brief description of the five disks:

Disk	Contents
CDTV Tools Volume 1 rev. 1	bookit, debox, keeper, includes, and more
CDTV Tools Volume II rev. 3	CDSetmap, autodocs, endrun, videodisc
CDXL Toolkit vers. 1.1	tools to create XL video streams
CDXL Toolkit "Sound on Film" example	sample XL streams
ISO DevPak Version 4.1	ISO mastering software

Further documentation is included on the disks.

I would particularly like to draw your attention to the CDXL Toolkit. This set of utilities lets you automate the creation of "XL movies". With a framegrabber, an ARexx script, an audio digitizer, the toolkit, and The Art Department Professional from ASDG, you can easily transfer video sequences from tape or videodisc to CD-ROM.

These disks replace any previous versions of these tools you might have received. Please remember that these tools are confidential. You are requested not to duplicate or distribute them to any third parties.

We hope you find these tools useful in creating exciting new CDTV titles. Comments and suggestions are welcome. Please use the report program located in the Debugtools directory of the Tools Vol. II disk.

Best regards,

BPhister

Benjamin Phister
Applications Manager

CDTV

CDXL TOOLKIT DEVELOPER DOCUMENTATION

Version 1.0 – October 1991

Overview

Warning – Read This
- *Safety*
- *Performance*

Commands
- *XLMake*
- *XLPlay*
- *XLInfo*
- *XLJoin*
- *XLCopy*
- *XLTrim*
- *XLAudio*
- *XLClean*

Creating Video

Adding Audio

Custom Programs

From PANTARAY, Inc. Ukiah, CA
Software by Ken Yeast
Produced by Carl Sassenrath

© 1991 Commodore Electronics, Ltd.
All Rights Reserved.

Overview

This package contains the first generation of CDXL tools to help you edit video/audio sequences and write programs to play the results. The distribution disks contain:

- Eight separate tool programs,
- A standard 'C' include file,
- An example source file,
- Example files in CDXL format.

Before using the tool programs, be certain to take the precautions stated in the next section.

The CDXL tools provided are:

XLMake creates or appends to a CDXL file. This tool accepts one or more IFF files (in the desired resolution and color mode), converts them to the standard CDXL format, and joins them together in sequence. Normally you would run this program as part of your video building script.

XLPlay simulates the playback of a CDXL file with approximate CDTV timing. XLPlay provides a simple means of viewing and hearing CDXL sequences on a standard Amiga computer as if they were on CDTV.

XLInfo displays detailed information about a CDXL file. The primary purpose of this tool is to provide size and timing information.

XLJoin combines multiple CDXL files into a single sequence. It can also insert a CDXL sequence into a second file at a particular point.

XLCopy copies frames from any point in a CDXL file and transfers them to a new file.

XLTrim removes frames from a CDXL file. Frames can be removed from any part of the file, or a file can be shortened overall by a specified amount.

XLAudio inserts an audio track into a CDXL file, ie., Sound on Film. The audio will become an integral part of the CDXL sequence and will be heard when you run XLPlay.

XLClean rebuilds the frame sequence numbers and linkage information within a CDXL file. This action is not normally required, but is sometimes useful if custom CDXL programs have been used.

Each of these tools will be described in detail shortly.

Warning – Read This

CDXL sequences can be massive. For each minute of CDXL, nine megabytes of disk space are required. The CDXL Toolkit programs perform operations that are very read/write intensive. Should your Amiga crash (power failure, etc.) during an important operation, your AmigaDOS disk structures may become invalid and require reformatting.

Special precautions must be taken. It is well advised that you edit CDXL files on a separate disk drive, or at the very least, a separate partition of your drive. This has two advantages:

Safety If your system crashes during CDXL editing, you may lose some of your work in the disk/partition, but the rest of your system and code will not be affected. If you keep your application source code on the same system, this is a very important consideration.

Performance After a few days/weeks of CDXL editing, your disk/partition will become fragmented. Such fragmentation will introduce seek delays into your editing operations, which will cause the tools to slow down, and may result in poor timing and glitches in the XLPlay command.

So these are good arguments for taking the time to get set up properly. Believe us, it will be worth the effort! ***Partition your disk or use another disk for editing!!!***

On our single disk systems, what we normally do is: put the standard workbench, libraries, devices, etc. into a small (6 meg) partition; put our development tools and code into a second (40 meg) partition; and use the rest of the disk for CDXL editing.

Commands

The CDXL Toolkit consists of a number of tools that operate as Shell (CLI) commands (they are not useful from the Workbench). Each of the tools require command arguments appropriate to its function. A brief summary of the arguments can be obtained by executing the command without any arguments.

Generally, each tool accepts both an input and an output file as arguments. The tool performs its function by reading the input file a frame at a time, making the required changes, and then writing to the output file. Normally, the input file is not modified (XLAudio has a special exception). This approach helps preserve your original files from accidental modification, but at the cost of extra disk space.

Note that the performance of the tools will vary depending on the speed of your Amiga processor and hard disk. The primary bottle neck, however, will usually be the AmigaDOS File System – large CDXL files may require significant time to process.

All of the tools deal with files that use a standard type of file format and header. Files that do not contain the proper header cannot be processed. This header is described later.

XLMake

XLMake is the primary tool for creating CDXL sequences. The purpose of this tool is to accept standard IFF files that are already in the desired resolution, color mode and depth, translate them into a raw bitplane format, and join them together as a sequence of frames to be read with CDXL. Each frame contains header information about its size, resolution, colors, and audio.

This tool is normally used in a script file or ARexx program during the capture of individual video frames; for example, you might use it along with the Art Department Professional (from ASDG). Each time a new frame has been captured, scaled, and color adapted, XLMake is invoked to combine the frame to the CDXL file under construction. To help illustrate its use, a useful AREXX script has been provided on the distribution disks.

For this tool to run you must have "iffparse.library" installed on your system (in LIBS:). (We provide a copy on the distribution disk for those who are unfamiliar with this library).

Summary

```
XLMake [options] <IFFFile1> [<IFFFile2>...] <File>
```

The IFF files are translated and appended to the File. If the File does not exist, it will be created. If it does exist, new frames are appended to the end.

Options

-a<n> allocate space for audio track. The value specified is the number of bytes to use per frame. It is not necessary to allocate the audio during XLMake. You can always add the audio later with XLAudio (see XLAudio for more information). This feature is included in XLMake so you can view playback with accurate timing before your audio has been added.

-d forces AVM (Advanced Video Mode) and DCTV. If you are storing AVM pictures, this flag must be set.

-f<n> frame sequence number (not strictly required, but handy to know). Normally you would be assigning each frame number for every frame added; however, frame numbers are not required and XLClean can be used to renumber files later. When multiple IFF files are specified, the frame number is incremented automatically.

-h forces HAM mode (only needed for ancient Amiga programs that didn't set the CAMG correctly).

Example

```
XLMake -f10 -d intro.xl

XLMake -a1024 intro2.xl
```

XLPlay

XLPlay is a very useful program that simulates the playback of a CDXL file with approximate CDTV timing. XLPlay provides a simple means of viewing and hearing CDXL sequences on a standard Amiga computer as if they were on CDTV. The purpose of this tool is to allow developers to view their CDXL files in order to make decisions and changes without remastering a CD (or creating an emulator version).

Playback timing is only approximate. The general timing characteristics are preserved (total play time), but the speed of AmigaDOS and the lack of CDXL on standard Amigas prevents precise timing from being achieved.

If an audio track is present, it will be played along with the video (mono only). Slight audio "drop–outs" will be heard due to the varying lengths of disk read times in the Amiga File System.

Summary

```
XLPlay [options] <File>
```

The specified File is played on the Amiga screen. The image will always be centered in the middle of the screen.

You can stop the play before it has finished by pressing the ESC key. To pause/unpause press the SPACE bar.

Options

-h force hi–res graphics mode (not normally required)

-i force interlace graphics mode (not normally required)

-r<n> repeat playback a number of times. This is helpful for short sequences that would be difficult to see.

-q quiet (mute the audio)

-s[<n>] show frame sequence numbers during playback. Handy if you need to know where to cut and edit. The optional numeric value indicates the "brightness" of the frame number display. It may range from 0 to 15, where 0 is black and 15 is white.

Example

```
XLPlay -r4 -s intro.xl
```

XLInfo

XLInfo displays detailed information about a CDXL file. The primary purpose of this tool is to provide accurate timing and other information for application developers.

This tool will calculate and display the total play time, number of frames, frame sizes, frame resolutions, number of colors, color modes, audio track size, audio period estimate, sequencing information, and out of sequence frame numbers.

Normally only the first frame's header is examined, and an estimate is calculated from its size and the total file size. This is done to save time. To check the entire file, use the –w option.

Summary

```
XLInfo [options] <File>
```

CDXL file information and statistics will be displayed for the specified File.

Options

–a[<n>] estimate audio sizes. This will produce an table of what various audio track sizes would produce in terms of timing, playback period, total audio size, etc. If the optional value is provided, it will be interpreted as the number of bytes per frame for audio, and its relevant information will be shown.

–f<n> first frame to examine,

–l<n> last frame to examine

The above two options use the ranges to examine only a section of a XL file. Information and statistics will only pertain to this section.

–w examine whole file. Normally XLInfo just looks at the first frame and estimates the rest. If you have a mixture of frame sizes or wish to analyze the entire file, use this option.

Example

```
XLInfo -w intro.xl
```

XLJoin

XLJoin combines multiple CDXL files into a single file. The purpose of this tool is to provide a means of splicing together separate video scenes to create a single CDXL video file.

XLJoin also allows the insertion of CDXL files anywhere within a CDXL file. This tool does not alter its input files in any way, thus preserving them for use in other video sequences.

Summary

```
XLJoin [options] <FileIn1> [<FileIn2>...] <FileOut>
```

Each of the input files will be appended (in the specified order) and written to the output file. Frames in the output file will have new sequence numbers.

Options

-i<n> insert into the first file at given frame. Normally XLJoin just combines a list of XL files into one XL file. This option allows you to "splice" the other files into the first input file (still producing the "out" file). This is useful when you wish to add an XL file into another "master" XL file.

Example

```
XLJoin intro.xl action.xl end.xl all.xl
```

XLCopy

XLCopy copies frames from any point in a CDXL file and transfers them to a new file. The purpose of this tool is to provide a simple means of extracting useful video sequences without altering the original file. This approach is easier and safer than XLTrim in some situations.

Summary `XLCopy [options] <FileIn> <FileOut>`

The specified portion of the input file will be copied to the output file. Frames in the output file will have new sequence numbers.

Options

–f<n> first frame to copy (inclusive). May be specified alone or in conjunction with the –l or –n options. When specified alone (without an end frame or length), frames through the end of the file are copied.

–l<n> last frame to copy (inclusive). May be specified alone or in conjunction with the –f or –n options. When specified alone (without a start frame or length), frames from the start of the file are copied. Using –n for a frame count will copy that many frames up to and including the last frame specified.

–n<n> number of frames to copy. This can be used with –f or -l but never with both.

Example

```
XLCopy -f10 -n100 intro.xl newintro.xl
```

XLTrim

XLTrim supplies a simple set of editing functions for removing selected frames from CDXL files. It is useful for removing unwanted video or reducing the play time to some maximum value.

Options are provided to: trim video frames from the beginning, middle, or end; remove every Nth frame; and trim the entire file down to a specified number of frames or playback time.

This program also has the capability of removing the audio track from a CDXL file.

Summary

```
XLTrim [options] <FileIn> <FileOut>
```

The specified portion of the input file will be copied to the output file. Frames in the output file will have new sequence numbers.

Options

Options that specify frame numbers are inclusive.

-a remove the audio track. Actually removes the track and frees that space from each frame.

-b<n> remove from beginning to this frame

-d<n> delete every <n>th frame

-e<n> remove from the end back this number of frames. This option assumes that the file contains equal sized frames.

-f<n> first frame to start removing

-l<n> last frame to remove

-n<n> number of frames to remove

-m<n> maximum number of frames desired for entire file. Frames will be removed at evenly spaced intervals throughout the file in order to cut the file to this size. This option assumes that the file contains equal sized frames.

-t<n> maximum play time in seconds. Frames will be removed at evenly spaced intervals throughout the file in order to cut the file to this size. This option assumes that the file contains equal sized frames.

Example

```
XLTrim -f10 -n10 intro.xl short.xl
      (removes frames 10-19)
XLTrim -m60 intro.xl intro60.xl
      (removes frames to make result 60 seconds)
XLTrim -e5 intro.xl introcut.xl
      (removes the last 5 frames)
```

XLAudio

XLAudio adds an audio sound track to an existing CDXL file. This operation is normally done as the last step in creating a CDXL audio/video sequence. The audio must already be adjusted to the correct sampling rate before laying down the track (see discussion in later section).

The tool will allocate the audio track for each CDXL frame if necessary, or overwrite the track if one already exists.

Only mono sound tracks are supported by this tool.

For this tool to run you must have "iffparse.library" installed on your system (in LIBS:). (We provide a copy on the distribution disk for those who are unfamiliar with this library).

Summary

```
XLAudio [options] <IFFSoundFile> <FileIn> [<FileOut>]
```

The IFF File is a sound file in either normal or compressed formats. FileIn holds the video sequence and may or may not already have an audio track. The output file contains both audio and video.

All frames in the input file must be of the same size.

Options

-a<n> allocate space for audio track. The value specified is the number of bytes to use per frame. See the discussion in a later section to determine the correct size. Also, the XLInfo –a option will print a table of possible values to consider using.

-i insert audio into existing track. This action operates on the input file and no output file is required. The frames in the input file must already have space allocated for the audio track (from XLMake or a previous XLAudio). This option is useful when the XL file is extremely large.

-e ignore audio size errors when inserting. When inserting audio with the -i option, XLAudio assumes that the audio track will remain the same size. When it does not, XLAudio complains unless this option is set.

-p preserve audio compression. The IFF sound file is copied with its compression intact.

Example

```
XLAudio -a1288 audio.iff video.xl final.xl
```

XLClean

XLClean rebuilds the frame sequence numbers and linkage information within a CDXL file. This action is not normally required, but is sometimes useful if custom CDXL programs have been used.

Summary `XLClean <FileIn> <FileOut>`

The input file is cleaned up and written to the output file.

Example `XLClean grab.xl video.xl`

Creating Video

To a large extent, the techniques you use for creating video will depend on the equipment you have at your disposal. A wide range of possibilities exist depending on your budget, ingenuity, and patience.

Of course the simplest solution is to directly generate the video graphics on your Amiga by using your favorite 2D painting or 3D modeling/rendering program. The only requirement is that the program be capable of producing displayable IFF files as output. Each frame would be stored as a separate file (on hard disk or on ram disk), and input to XLMake as needed.

Another approach might be to use a video frame digitizer that can be operated either manually or automatically (from AREXX or ADPRO). The images from this digitizer could be processed and altered with ADPRO then saved as an IFF file and again entered as input to XLMake. An simple AREXX script to do this might be:

```
/* CDXL ADPRO program for PPS Framegrabber: */
OPTIONS RESULTS
ADDRESS "ADPro"
LFORMAT "FRAMEGRABBER"
SFORMAT "IFF"
GETNUMBER '"How many frames?"'
if RC ~= 0 then exit
FRAMES = ADPRO_RESULT
DO I = 1 to FRAMES
      ADDRESS "ADPro"
      OKAY2 '"Ready to grab?"'
      if RC = 0 then exit
      LOAD "XXX"
      SCREEN_TYPE 0
      ABS_SCALE 160 100
      EXECUTE
      SAVE "RAM:TempFrame" "SCREEN"
      ADPRO_DISPLAY
      PAUSE 100
      ADPRO_UNDISPLAY

      ADDRESS Command
      XLMAKE "RAM:TempFrame" FILENAME
      END
```

Adding Audio

Suppose want to make a short video clip....

60 sec Total Run Time

14 K/sec audio quality (approx)

12168 bytes/frame (from XLInfo)

First determine what you know for sure:

Total Size:

60 sec * 150 K/sec * 1024 bytes/K = 9216000 bytes

Total Audio Size:

60 sec * 14092 bytes/sec = 845520 bytes

Calculate basic figures:

CDXL less audio:

9216000 - 845520 = 8370480 bytes

Number of frames:

8370480 bytes / 12168 bytes/frame = 688 frames

Finally, calculate:

Audio sample size per frame:

845520 bytes / 688 frames = 1228 bytes / frame

Now you can proceed as follows:

Use XLTrim to cut your grab down to 688 frames:

XLTrim -m688 video.grab video.688

With a program like "Audition4", resample your audio to 14K:

Resample to 14092 bytes/sec

Save file as audio.14092

Finally, merge the video and audio data into a single file:

XLAudio -a1228 audio.14092 video.688 final.xl

Now, see how it looks and sounds:

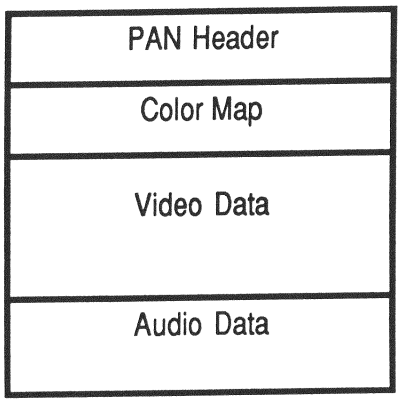
XLPlay final.xl

Keep in mind that there may be a few small timing errors when playing this on an Amiga. Normally you won't see these, but you may hear them. Unfortunately, the only way to be absolutely sure is to test with the emulator using your application program.

Custom Programs

Within an application you will need code to play the CDXL sequences that you have created. You may also want to create special tools for manipulating CDXL sequences during the authoring process. To do this, you must understand the structure of a CDXL file and the philosophy behind it.

Standard CDXL files are made up of one or more frames. The general format of a frame is:



When a file contains more than one frame, the frames are simply concatenated, header and all. That is, one frame follows immediately after another. The only restriction is that frames must be aligned on even byte (word aligned) boundaries.

Various portions of a frame are optional. Actually, the only required section is the PAN Header, which contains information about the frame. The Color Map, Video Data, and Audio Data may or may not exist.

The PAN Structure is key to your access of CDXL files. It is defined in the file "pan.h" as:

```
struct PanFrame
{
   UBYTE Type;
```

This indicates what type of structure you are reading. Currently this field must be set to PAN_STANDARD.

```
   UBYTE Info;
```

This field indicates variations on the general Type. Various options are discussed below.

```
   ULONG Size;
```

This is the total size of the frame in bytes. It includes the entire PanFrame structure, color map, video, and audio. With this value, you can skip to the next frame.

```
ULONG Back;
```

This is the offset from the start of this frame back to the beginning of the previous frame. While CDXL cannot play backwards, this field is useful for tools.

```
ULONG Frame;
```

The Frame Sequence Number. First frame is number one (not zero). This number uniquely identifies the frame within a file. A zero value indicates that the number is undefined, and tools should print out a "??????" in the number field to show this.

```
UWORD XSize;
```

The width of the video in pixels.

```
UWORD YSize;
```

The height of the video in lines.

```
UBYTE Reserved;
```

Reserved for future. CURRENTLY, MUST BE ZERO!

```
UBYTE PixelSize;
```

The "depth" of a pixel. That is, the number of bits used to indicate a color (with or without a color map). A zero value indicates that no video data exists within the frame. Example values: 4 for 16 color, 6 for HAM, 24 for RGB24, etc.

```
UWORD ColorMapSize;
```

The total size of the color map in bytes. If you require more than 64K of color map, then you shouldn't be using CDXL (but you can use your own custom CDXL file, see below). A zero value indicates no color map. Typical values are 32 for 16 colors, 64 for 32 colors, 64 for half-bright (64 color) mode, 32 for HAM.

```
UWORD AudioSize;
```

Total size of the audio sample area in bytes. A zero indicates no audio is present. Total size must include all channels (when multiple audio channels are used).

```
UBYTE PadBytes[8];
```

This area is reserved for future use. For now it must always be set to zero.

```
};
```

The PAN Structure is optionally followed by the Color Map, Video data, and Audio data. Each of these areas must be even byte (16 bit word) aligned.

The Type field defined above is set to indicate the type of header structure being used. Over time, CDXL will evolve and this type will be critical for identifying what type of frame we are about to process. Right now the valid possibilities for this field are:

PAN_STANDARD – is used to indicate that the frame is in a standard format and that the above structure is used.

PAN_CUSTOM – is provided for you to create your own formats, but without causing problems for general tools. Custom formats allow you to lay your data however you need it. The only requirement is that the file start with the structure:

```
struct PanHead
{
    UBYTE Type;
```

Set to PAN_CUSTOM to identify this as nonstandard.

```
    UBYTE Info;
```

Reserved. Set to zero.

```
    ULONG Size;
```

This is the total size of the custom frame in bytes, including this structure. It is used to jump from the beginning of this frame to the beginning of the next frame.

```
};
```

PAN_SPECIAL – is reserved for experimental, and proposed formats.

All other values for the Type field are reserved for new frame structures. As CDXL evolves, new PAN types will be published.

In the PanFrame structure the info field describes various formats for the data contained within the frame.

PIV_MASK defines a bit field that tells you what video decoding should be used to display the image. Choices are:

PIV_STANDARD – for normal encoding (Amiga, RGB).

PIV_HAM – for hold and modify encoding

PIV_YUV – for luma-chroma encoding (eg. 844 bit YUV)

PIV_AVM – for CDTV advanced video mode and DCTV

Commodore will add new values to this as needed. Pixel value orientation is NOT specified here. It is defined in a separate bit field, PIF_MASK, that indicates the orientation format:

PIF_PLANES – data is interleaved on a bitplane basis, eg. in the normal Amiga bitplane style. Each plane contains one bit of the pixel value.

PIF_PIXELS – data is interleaved on per pixel basis, eg. the chunky style like that used for VGA data, 8 bits per pixel, 24 bits per pixel, whatever.

PIF_LINES – data is interleaved on a line basis. This can be thought of as each line being a single bitplane to itself.

Another field, PIA_MASK indicates audio options. It tells you whether the audio is mono or stereo.

A few size macros are provided to help you move through a file. These should be used as needed.

CDTV Developer Notes

Using CDXL

CDTV Transfer Lists

Version 1.0

By Carl Sassenrath

Confidential

Patent Pending

Copyright © 1991 Commodore Electronics, Ltd.

1. THE PROBLEM: SEEK TIME

The total transfer time for a given group of CD data blocks includes not only the time required to transfer the data into memory, but also the delay incurred in reaching the desired blocks on disk. In this document the former will be called the *data transfer time* and the latter will be termed the *seek time*.

The data transfer time for a typical CD-ROM read operation depends on the drive's frame rate and the disk's sector size. The current technology for CD mechanisms offers a frame rate of 75 f/s (frames/second). This is the rate required to produce full quality CD-DA (digital audio). The sector size used for holding a frame is approximately 2k bytes, but it may increase a few percent when complete error correction is not required. From these numbers a theoretical *data transfer rate* can be calculated:

```
75f/s * 2k/f  = 150k/s
```

Typical CD drive specifications state that the maximum seek time is 0.8 seconds, with an average time of 0.5 seconds. You might falsely assume from such documents that the minimum seek time is zero. This is rarely, if ever, the case. It is not widely recognized nor advertised that the minimum seek time can vary from around 0.02 to 0.35 seconds. Rarely is the laser precisely where it needs to be – directly before the desired starting point.

The total transfer time for a read is calculated by dividing the transfer rate by the quantity of data and adding in the seek time. A 48k read with an average seek time would take:

```
(48k / 150k/s) + 0.5s = 0.82s
```

Notice that 61% of the total time is spent in the seek alone! Another way of thinking about this loss is to relate seek times to data sizes. An average seek is equivalent to the transfer of:

```
150k/s * 0.5s = 75k
```

If this is for a single transfer, think about a program which contains many such transfers. The loss accumulates and greatly degrades the performance of the program. As an example, suppose your program needs ten color HAM images and 20 seconds of stereo Amiga sound. Let's also suppose the data is clustered together on disk, reducing the seek time to 0.25 seconds:

```
Size of each HAM image = 48k
Size of each second of stereo sound = 32k
((48k / 150k/s) + 0.25s) * 10 = 5.7s
((32k / 150k/s) + 0.25s) * 20 = 9.3s
Total transfer time = 15s
Total data transfered = 1120k
Total data rate = 75k/s
```

So CD-ROM should not be thought of as a truly random access medium. Half of your total transfer time has been consumed by seeking – a very high cost even when data is organized properly on disk.

2. THE SOLUTION: DON'T SEEK

Considering the penalty for seeking, the only clear solution is not to seek. Read as much data as possible in a single read operation, even if this means dumping some of the data. Remember the data transfer cost of an average seek is about 75k, so you can afford to dump a lot of data and still be ahead.

Programmer's will need to rethink their strategies for data access. For example, they should duplicate sections of data rather than seek backward on the disk; put a large table of values on disk every N seconds if you cannot spare the memory space; build special hashed lookup tables and cache them in memory; or create alternate index tables on disk at development time, rather than chaining through sectors at run time.

Using the above program example, if a single read were achievable, only one seek would be required, and this would result in an effective transfer of nearly twice the speed:

```
(48k / 150k/s) * 10 = 3.2s
(32k / 150k/s) * 20 = 4.3s
Total transfer time = 7.5s + 0.5s = 8s
Total data transfered = 1120k
Total data rate = 140k/s
```

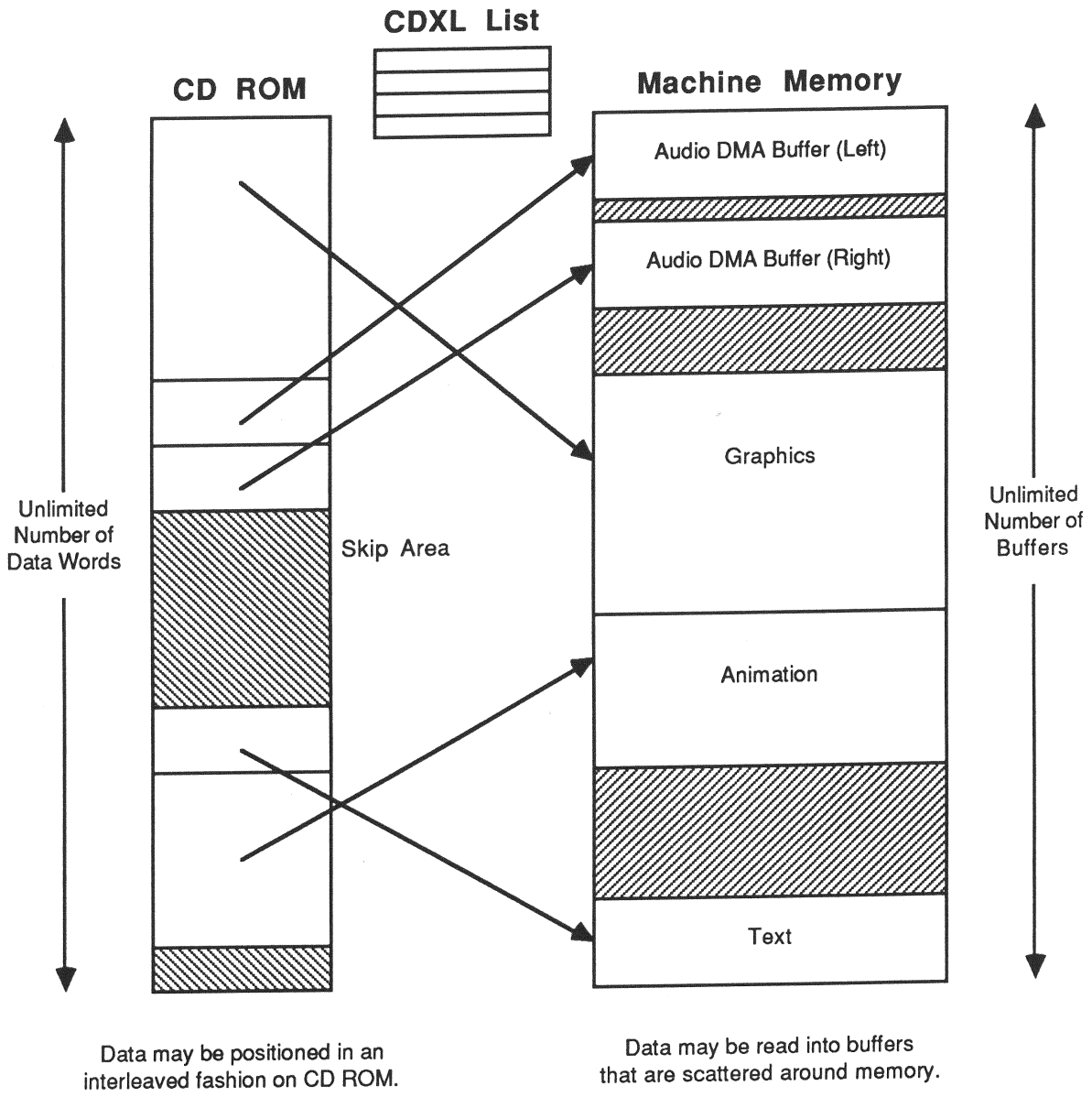
Of course this approach requires data to be organized on sequential sectors of the disk.

3. CDXL

Initially the "no seek" solution may not seem practical. Programs often need to read several different types of data into memory: graphics, sound, music, text, code, etc. All of these may be of different sizes and have different target addresses. If only a single read can be performed, data would need to be read from CD into an intermediate memory buffer and distributed to other locations from there. This would reduce system efficiency by moving data twice, and it would also consume a substantial amount of memory for the intermediate buffer.

Now, if the CD device driver itself had the capability of distributing data into multiple buffers other overhead would be eliminated – data would not be moved twice and the intermediate buffer would be eliminated. CD data of any length could be placed anywhere in memory with a single high-speed DMA transfer.

The CDTV device driver provides this capability through the use of CD transfer lists (CDXL). These lists allow programmers to specify the size and placement of data during a continuous read operation. Data can be distributed around memory into graphic displays, sound buffers, text tables, or whatever, without additional seek operations. While it is clear that this mode of operation may not fit all cases, it does benefit many types of multimedia applications, and is generally useful when high throughput is desired.



The transfer is performed as a single read from CD-ROM.

4. BASIC OPERATION

Within the CDTV device driver CDXL is supported through a special read command CD_READXL. This command, like the CD_READ command, begins a read operation from the drive by seeking to the desired location, but once the data is ready CD_READXL begins traversing a linked list (the *transfer list*) to determine the destinations for the data. The field format for the CD_READXL command is:

```
ior.io_Command = CD_READXL;
ior.io_Offset = SectorNumber;
ior.io_Length = SectorCount;
ior.io_Data = XferList;
```

The **io_Offset** field contains the starting position for the read as a logical sector number within the CD-ROM track (unlike the CD_READ command).

The **io_Length** field limits the total number of sectors to be read. You can calculate this number based on the number of bytes to be transferred in the list, or you can specify a -1 to indicate no limit. In the latter case, when the transfer list has completed, the drive operation will be aborted. When a zero length is used, it produces a seek effect and no data is read.

The **io_Data** field points to the transfer list – a standard *Amiga Exec* style **MinList** that contains any number of separate **MinNodes** describing the transfer of data. The pointer is not limited to a list header. It may point to any node within the list, so partial lists or circular lists can be executed.

Each node in a transfer list contains information regarding a destination for a portion of the data to be read. There is one of these nodes for each separate data buffer. The *devices/cd.h* file contains the key structure:

```
struct CDXL
{
        struct MinNode *Node;
        UWORD *Buffer;
        ULONG Length;
        FUNC  DoneCode;
        ULONG Actual;
};
```

The **MinNode** structure is used for list linkage by the Exec list functions. You can use **AddHead()**, **AddTail()**, **RemHead()**, **RemTail()**, **Insert()**, and **Remove()** to construct the list.

The **Buffer** pointer determines the location to store data being read. It must be word aligned (an even number). This buffer is filled with the number of bytes specified in the **Length** field, and the length must also be an even number (because it is two times the number of words).

The buffer pointer may optionally be zero. In such a case the data is read from CD, but discarded into a bit bucket. This feature is quite useful for skipping unused areas of the CD without terminating the continuous read operation.

As soon as the specified number of bytes have been read, the **DoneCode** function callback is executed. Your callback function is executed as part of interrupt handling. It is running in supervisor mode on the supervisor stack and all of the normal interrupt code restrictions apply. You should keep your code short and to the point – signal a task, send a message, set a flag, or cause a software interrupt. If you take too long, you may cause the CD DMA

channel to timeout. More about these functions later. Also, if a zero **Length** was specified, no transfer will occur and your callback will happen immediately.

Finally, the **Actual** field indicates the number of bytes read into your buffer. Normally, this is the same as **Length**, unless an error occurred or you are examining it before the read has finished.

Once the transfer list has been built, the **DoIO()** or **SendIO()** functions are used to start the transfer.

5. TRANSFER LIST DETAILS

The list approach was taken to give complete flexibility in ordering and locations for data buffers. Transfer lists can be built anywhere in memory – they need not be next to or part of the buffers. Because your code links the list together at run time, you can create and modify the order of nodes to suit your own requirements.

The precise form of a transfer list will depend entirely on how you wish to partition your data and what buffers it must be distributed to. When creating the CD, special attention must be paid to the organization of sectors used with CDXL. You will need to consider not only the sizes and locations, but also the timing of relationships of the data.

As a introductory CDXL example, suppose we have a program with data requirements in the order:

48k HAM Image

20k Stereo Sound

From C you might build the elements for a transfer list from static structures:

```
struct MinList XferList;
struct CDXL ImageXL =
{
        NULL,NULL,          /* Linkage   */
        ImageBuf,           /* Buffer    */
        48*1024,            /* Length    */
        NULL,               /* DoneCode  */
        0                   /* Actual    */
};
struct CDXL SoundXL =
{
        NULL,NULL,          /* Linkage   */
        SoundBuf,           /* Buffer    */
        20*2024,            /* Length    */
        NULL,               /* DoneCode  */
        0                   /* Actual    */
};
```

The structures would then be linked together with:

```
NewList(&XferList);
AddTail(&XferList,&ImageXL);
AddTail(&XferList,&SoundXL);
```

A generally more useful approach would be to build a function for creating new list elements:

```
struct CDXL *NewXL(lst,buf,len,code)
        struct MinList *lst;
        UWORD *buf;
        ULONG len;
        FUNC code;
{
        struct CDXL *x;
        x = malloc(sizeof(struct CDXL));
        if (x == NULL) Error(NO_MEM);
        x->Buffer = buf;
        x->Length = len;
        x->DoneCode = code;
        AddTail(lst,x);
        return x;
}
```

The new initialization would become:

```
NewList(&XferList);
NewXL(&XferList,ImageBuf,48*1024,NULL);
NewXL(&XferList,SoundBuf,20*1024,NULL);
```

and adding new elements would be as simple as adding a new line. Of course for both of these structures your CD would need to be laid out accordingly.

As another example, suppose you have 40 HAM images that you would like to display as quickly as possible. Ignoring compression techniques for now, each image would take 48k, which means you could read 3.125 images per second. It is assumed that your CD has been mastered so that all 40 images are sequentially positioned within the same file (or alternately when images are mod 2048, you have 40 separate files positioned adjacent to one another).

In addition, you would like to double buffer these images – reading the next image while the current image is displayed. To do this you setup two buffers, and read alternate images into alternate buffers.

Finally, you build the transfer list. Creating it in a normal fashion would require 40 entries, 20 for each buffer. There is, however, an easy trick that will reduce this effort. You may create two entries, one for each buffer and make the list circular. To do this a new function will be defined:

```
LoopList(lst)
        struct List *lst;
{
        struct Node *head = lst->lh_Head;
        struct Node *tail = lst->lh_Tail;
        if (head->ln_Succ == NULL) return; /* empty */
        head->ln_Pred = tail;
        tail->ln_Succ = head;
}
```

This requires an understand of Exec lists, which can be gained from the RKM or other sources. Note that once this function has been executed, do not use the AddXXXX() and RemXXXX() functions on the list. **Insert()** and **Remove()** however should work properly. Your code now becomes:

```
NewList(&XferList);
NewXL(&XferList,ImageBuf1,48*1024,NULL);
NewXL(&XferList,ImageBuf2,48*1024,NULL);
LoopList(&XferList);
```

When the read command is performed, the device driver will loop endlessly through this list, filling one buffer, then the other. The loop is broken when the driver has read all sectors specified in the io_Length field, or an AbortIO has occurred. It is also likely that you will be providing some form of callback function, which could count the reads and also break the loop.

6. DONECODE CALLBACK FUNCTIONS

CDXL is supported by a custom Amiga/CDTV DMA chip. Because of this, read operations take relatively little CPU horsepower and CDXL operations occur transparently[1] to most tasks running on the system. For simpler types of read operations, you might be able to wait for the I/O request message to return as a sign that your command has finished. However, in more complex cases of reading, you may want detailed knowledge of which buffers have been filled.

As mentioned above, each transfer list entry may contain a function pointer to be called that entry has been executed. This feature is tremendously valuable for most types of CD data dependent programs.

---

[1] The DMA does steal CPU cycles every so often, which could slow code down some.